Prof. Dr. Oscar Nierstrasz
Pascal Gadient, Leonel Merino

Solution
Series 08 — 08.11.2017 – v1.0
Condition Objects

Exercise 1: General Questions (2 Points)

Answer the following questions:

a) Why are servers (e.g. web servers) usually structured as thread-per-message gateways? **Answer:**

Servers (e.g. web servers) usually must handle many requests, some of them quite resource-intensive. Therefore, it makes sense to let each request (service) to be handled by a separate helper thread, as long as the overhead of thread construction does not exceed the helper costs. Load balancing can be realized by distributing different tasks among separate threads or processes.

b) What are condition objects? Name at least one advantage and one disadvantage of using condition objects. **Answer:**

Condition objects are a programming pattern used to encapsulate waiting and notification strategies in the context of guarded methods.

Advantages:

- *More complicated synchronization strategies or scheduling policies can be sourced out to simplify the class design*
- *By isolating conditions, one can often avoid notifying waiting threads that possibly cannot proceed given a particular state change (efficiency)*

Disadavantages:

- *The first advantage may turn into a disadvantage: Java language constraints might increase design complexity (e.g. see solution of bounded counter example in the lecture)*
- *It may lead to "design overhead" in some situations*
- *If used without care, it may easily leads to nested monitor problems (but this applies to synchronization mechanisms in general)*

c) Why does the *SimpleConditionObject* from the lecture not need any instance variables? **Answer:**

The Condition *class does not need to implement any state, rather it specifies synchronization behaviour. Simple condition objects are then used as "synchronization variables" by other classes (e.g. bounded counter) that need synchronization.*

d) What are "permits" and "latches"? When it is natural to use them? **Answer:**

Basically, permits are special condition objects that behave similarly to counting semaphores. A permit class maintains a permit counter representing the number of available "permits" (e.g. number of allowed objects in bounded buffer, number of possible increments in bounded counter, etc.).

In fact, one could say that a permit is a counting semaphore implemented as condition object. The signal operation increments, and the await operation decrements the permit counter. A latch is a condition that is initially false, but once set to true, it remains true. See for example the CountDownLatch *class in* java.util.concurrent.
These kinds of mechanisms are useful whenever synchronization depends on the value of some counter.

Exercise 2: Questions about Futures (3 points)

Consider the sample code *FutureTaskDemo.java* and *EarlyReplyDemo.java* which you both find in *Series-08-ConditionObjects-Code.zip*[1] in the package *asynchrony*. In both cases, several client threads request a server to compute fibonacci numbers.

a) Which implementation would you prefer for this kind of problem? Is there any considerable difference at all? Justify your answer! **Answer:**

In the early-reply approach, the client waits for a service (which has been delegated by the host to a helper) until the host gets the result of the computation (the host retains synchronization). In the future-task approach, the client receives from the host a future object, which represents the (future) result of the (asynchronous) computation. This computation executes in parallel while the client in fact may continue its work.
In our example, every demoThread receives a future object, and depending on the exact scheduling, the clients waiting for the faster computations are served first. In the early reply example, all clients are waiting for the slowest computation, if this was the first request. It depends much on the order of requests which clients are served first. So the future-based solution seems more flexible in this case.

b) Write a new class *FutureTaskExecDemo.java* that uses an executor service to compute the future task and to execute the clients, instead of creating explicit new threads. What is the benefit of using executors? **Answer:**

The task is usually submitted to a pool of threads of which one will execute your task. This is convenient because you don't have to create explicitly new threads, and the code that submits the task does not have to be aware of which thread exactly executes the task.

c) Add a time constraint such that the client thread waits for at most a given amount of time for the result. **Answer:**

The solution can be found in FutureTaskExecDemo.java *in the* Series-08-ConditionObjects-Code-Solution.zip[2] *folder.*

[1] http://scg.unibe.ch/download/lectures/cp-exercises-2017/Series-08-ConditionObjects-Code.zip
[2] http://scg.unibe.ch/download/lectures/cp-exercises-2017/Series-08-ConditionObjects-Code-Solution.zip

Exercise 3: Nested Monitor (2 points)

Farmer Napoleon owns a magic chicken called Clarissa who is supposed to lay infinitely many eggs. Napoleon has hoped to dispose an endless source of eggs to build up his egg-imperium. But there is a serious deadlock problem hidden behind the story. Your task is to resolve this problem in order to let Napoleon continously retrieve eggs from Clarissa. Be careful that your solution is data race free.

You can find the code in the file *CP-Series8.zip* of the previous task in the package *eggFarm*.
Answer:

You can find the code in the file CP-Series8-Sol.zip *of the previous task in the package* eggFarm.

Exercise 4: Thread Speed Evaluation (3 points)

We provide you with an implementation of a Pi approximation tool. This tool uses the Leibniz formula for Pi with infinite series for approximation. Your task is to observe the characteristics of the threads as well as their calculation speed. You can simply modify the values of the variables *concurrentThreads* and *rounds* in the class *ThreadedPiCalculator* to change the amount of allowed concurrent threads, respectively the accuracy of the result, i.e., number of iterations performed. It is very important that you perform multiple runs before answering the questions below, because the outputs each run can vary up to a degree. You can find the sample code on GitHub at `https://github.com/pgadient/concurrency_e08t04.git`. Please answer the following questions:

a) What amount of processing cores does the CPU in your notebook have and what's the model / manufacturer of it? **Answer:**

Intel Core i7-4600 CPU, providing 2 physical and 4 logical cores.

b) Does the implementation scale well, i.e., more concurrent threads help greatly to reduce the overall calculation time? Please provide concrete runtimes you experienced! **Answer:**

No, it does not scale well. With just one concurrent thread we achieve runtimes for 10,000 iterations on a dual-core notebook between 950 and 3,500 ms, whereas with two concurrent threads we achieve almost identical runtimes. If it would scale nicely, two threads would only use 50% of the time one thread requires.

c) Depending on your results, why or why not does the solution scale well? **Answer:**

It does not scale well, because the implementation produces way too much overhead with the thread management. The core issue here is that the desired calculations per thread are much less complex than the actual operations needed for instantiation and maintenance of the threads. In other words, more CPU cycles are used for the thread management than the actual calculation itself.

d) How would you improve the runtime with respect to faster calculations (without changing the algorithm)? **Answer:**

We could group individual operations into much larger chunks and calculate these chunks on

different threads. Then each thread would work for several seconds or minutes which would be beneficial.

e) Which algorithm would you recommend as drop-in replacement for the Leibniz formula for faster calculation? **Answer:**

There exist many different algorithms. One of the more promising approaches that supports quite a high degree of parallelism is called "BaileyBorweinPlouffe" formula. You can find more information on `https://en.wikipedia.org/wiki/Bailey%E2%80%93Borwein%E2%80%93Plouffe_formula`

f) Why do the runtimes with identical parameters vary so much? **Answer:**

The Java VM performs various internal optimizations (caching, command reordering, ...) and relies on external threading APIs of the underlaying operating system, that may not perform identical on each run. Another influential factor are other concurrent processes that work concurrently on the system.